Chapter 5

Discrete Time: MATLAB Programs

This chapter is dedicated to illustrating the examples, theory, and algorithms, as presented in the previous chapters, through a few short and easy-to-follow MATLAB programs. These programs are provided for two reasons: (i) For some readers, they will form the best route by which to appreciate the details of the examples, theory, and algorithms we describe; (ii) for other readers, they will be a useful starting point to develop their own codes. While ours are not necessarily the optimal implementations of the algorithms discussed in these notes, they have been structured to be simple to understand, to modify, and to extend. In particular, the code may be readily extended to solve problems more complex than those described in Examples 2.1–2.7, which we will use for most of our illustrations. The chapter is divided into three sections, corresponding to programs relevant to each of the preceding three chapters.

Before getting into details, we highlight a few principles that have been adopted in the programs and in the accompanying text of this chapter. First, notation is consistent between programs, and it matches the text in the previous sections of the book as far as possible. Second, since many of the elements of the individual programs are repeated, they will be described in detail only in the text corresponding to the program in which they first appear; the short annotations explaining them will, however, be repeated within the programs. Third, the reader is advised to use the documentation available at the command line for *any* built-in functions of MATLAB; this information can be accessed using the `help` command—for example, the documentation for the command `help` can be accessed by typing `help help`.

5.1 Chapter 2 Programs

The programs `p1.m` and `p2.m` used to generate the figures in Chapter 2 are presented in this section. Thus these algorithms simply solve the dynamical system (2.1) and process the resulting data.

Electronic supplementary material The online version of this chapter (doi: 10.1007/978-3-319-20325-6_5) contains supplementary material, which is available to authorized users.

K. Law et al., *Data Assimilation*, Texts in Applied Mathematics 62,
DOI 10.1007/978-3-319-20325-6_5

5.1.1. p1.m

The first program, `p1.m`, illustrates how to obtain sample paths from equations (2.1) and (2.3). In particular, the program simulates sample paths of the equation

$$u_{j+1} = \alpha \sin(u_j) + \xi_j, \tag{5.1}$$

with $\xi_j \sim N(0, \sigma^2)$ i.i.d. and $\alpha = 2.5$, both for deterministic ($\sigma = 0$) and stochastic dynamics ($\sigma \neq 0$) corresponding to Example 2.3. In line 5, the variable `J` is defined, which corresponds to the number of forward steps that we will take. The parameters α and σ are set in lines 6–7. The seed for the random-number generator is set to `sd`$\in \mathbb{N}$ in line 8 using the command `rng(sd)`. This guarantees that the results will be reproduced exactly by running the program with this same `sd`. Different choices of `sd`$\in \mathbb{N}$ will lead to different streams of random numbers used in the program, which may also be desirable in order to observe the effects of different random numbers on the output. The command `sd` will be called in the preamble of all of the programs that follow. In line 9, two vectors of length `J` are created, named `v` and `vnoise`; after the program has run, these two vectors contain the solutions for the case of deterministic ($\sigma = 0$) and stochastic dynamics ($\sigma = 0.25$) respectively. After the initial conditions are set in line 10, the desired map is iterated, without and with noise, in lines 12–15. Note that the only difference between the forward iteration of `v` and that of `vnoise` is the presence of the `sigma*randn` term, which corresponds to the generation of a random variable sampled from $N(0, \sigma^2)$. Lines 17–18 contain code that graphs the trajectories, with and without noise, to produce Figure 2.3. Figures 2.1, 2.2, and 2.5 were obtained by simply modifying lines 12–15 of this program, in order to create sample paths for the corresponding Ψ for the other three examples; furthermore, Figure 2.4a was generated from output of this program, and Figure 2.4b was generated from output of a modification of this program.

```
 1  clear;set(0,'defaultaxesfontsize',20);format long
 2  %%% p1.m - behaviour of sin map (Ex. 1.3)
 3  %%% with and without observational noise
 4
 5  J=10000;% number of steps
 6  alpha=2.5;% dynamics determined by alpha
 7  sigma=0.25;% dynamics noise variance is sigma^2
 8  sd=1;rng(sd);% choose random number seed
 9  v=zeros(J+1,1);  vnoise=zeros(J+1,1);% preallocate space
10  v(1)=1;vnoise(1)=1;% initial conditions
11
12  for i=1:J
13      v(i+1)=alpha*sin(v(i));
14      vnoise(i+1)=alpha*sin(vnoise(i))+sigma*randn;
15  end
16
17  figure(1), plot([0:1:J],v),
18  figure(2), plot([0:1:J],vnoise),
```

5.1.2. p2.m

The second program presented here, `p2.m`, is designed to visualize the posterior distribution in the case of one-dimensional deterministic dynamics. For clarity, the program is separated into three main sections. The `setup` section in lines 5–10 defines the parameters of the problem. The model parameter `r` is defined in line 6, and it determines the dynamics of the forward model, in this case given by the logistic map (2.9):

$$v_{j+1} = rv_j(1 - v_j). \tag{5.2}$$

The dynamics are taken as deterministic, so the parameter `sigma` does not feature here. The parameter `r` is equal to 2, so that the dynamics are not chaotic, as the explicit solution given in Example 2.4 shows. The parameters `m0` and `C0` define the mean and covariance of the prior distribution $v_0 \sim N(m_0, C_0)$, while `gamma` defines the observational noise $\eta_j \sim N(0, \gamma^2)$.

The `truth` section in lines 14–20 generates the true reference trajectory (or truth) `vt` in line 18 given by (5.2), as well as the observations `y` in line 19 given by

$$y_j = v_j + \eta_j. \tag{5.3}$$

Note that the index of `y(:,j)` corresponds to observation of `H*v(:,j+1)`. This is due to the fact that the first index of an array in MATLAB is `j=1`, while the initial condition is v_0, and the first observation is of v_1. So effectively the indices of `y` are correct as corresponding to the text and equation (5.3), but the indices of `v` are off by 1. The memory for these vectors is *preallocated* in line 14. This is unnecessary, because MATLAB would simply *dynamically allocate* the memory in its absence, but it would slow down the computations due to the necessity of allocating new memory each time the given array changes size. Commenting this line out allows observation of this effect, which becomes significant when `J` becomes sufficiently large.

The `solution` section after line 24 computes the solution, in this case the pointwise representation of the posterior smoothing distribution on the scalar initial condition. The pointwise values of the initial condition are given by the vector `v0` (v_0) defined in line 24. There are many ways to construct such vectors, and this convention defines the initial (0.01) and final (0.99) values and a uniform step size 0.0005. It is also possible to use the command `v0=linspace(0.01,0.99,1961)`, defining the *number* 1961 of intermediate points, rather than the step size 0.0005. The corresponding vector of values of `Phidet` (Φ_{det}), `Jdet` (J_{det}), and `Idet` (I_{det}) are computed in lines 32, 29, and 34 for each value of `v0`, as related by the equation

$$\mathsf{I}_{\text{det}}(v_0; y) = \mathsf{J}_{\text{det}}(v_0) + \Phi_{\text{det}}(v_0; y), \tag{5.4}$$

where $\mathsf{J}_{\text{det}}(v_0)$ is the **background** penalization and $\Phi_{\text{det}}(v_0; y)$ is the **model–data misfit** functional given by (2.29b) and (2.29c) respectively. The function $\mathsf{I}_{\text{det}}(v_0; y)$ is the negative log-posterior as given in Theorem 2.11. Having obtained $\mathsf{I}_{\text{det}}(v_0; y)$, we calculate $\mathbb{P}(v_0|y)$ in lines 37–38, using the formula

$$\mathbb{P}(v_0|y) = \frac{\exp(-\mathsf{I}_{\text{det}}(v_0; y))}{\int \exp(-\mathsf{I}_{\text{det}}(v_0; y))}. \tag{5.5}$$

The trajectory v corresponding to the given value of v_0 (`v0(i)`) is denoted by `vv` and is replaced for each new value of `v0(i)` in lines 29 and 31, since it is required only to compute `Idet`. The command `trapz(v0,exp(-Idet))` in line 37 approximates the denominator of the above by the trapezoidal rule, i.e., the summation

$$\texttt{trapz}(\texttt{v0}, \texttt{exp}(-\texttt{Idet})) = \sum_{i=1}^{N-1} (\texttt{v0}(\texttt{i}+1) - \texttt{v0}(\texttt{i})) * (\texttt{Idet}(\texttt{i}+1) + \texttt{Idet}(\texttt{i}))/2. \qquad (5.6)$$

The rest of the program deals with plotting our results, and in this instance, it coincides with the output of Figure 2.11b. Again, simple modifications of this program were used to produce Figures 2.10, 2.12, and 2.13. Note that `rng(sd)` in line 8 allows us to use the same random numbers every time the file is executed; those random numbers are generated with the seed `sd`, as described in Section 5.1.1. Commenting this line out would result in the creation of new realizations of the random data y, different from those used to obtain Figure 2.11b.

```
1  clear; set(0,'defaultaxesfontsize',20); format long
2  %%% p2.m smoothing problem for the deterministic logistic map (Ex. 1.4)
3  %% setup
4
5  J=1000;% number of steps
6  r=2;% dynamics determined by r
7  gamma=0.1;% observational noise variance is gamma^2
8  C0=0.01;% prior initial condition variance
9  m0=0.7;% prior initial condition mean
10 sd=1;rng(sd);% choose random number seed
11
12 %% truth
13
14 vt=zeros(J+1,1); y=zeros(J,1);% preallocate space to save time
15 vt(1)=0.1;% truth initial condition
16 for j=1:J
17     % can be replaced by Psi for each problem
18     vt(j+1)=r*vt(j)*(1-vt(j));% create truth
19     y(j)=vt(j+1)+gamma*randn;% create data
20 end
21
22 %% solution
23
24 v0=[0.01:0.0005:0.99];% construct vector of different initial data
25 Phidet=zeros(length(v0),1);Idet=Phidet;Jdet=Phidet;% preallocate space
26 vv=zeros(J,1);% preallocate space
27 % loop through initial conditions vv0, and compute log posterior I0(vv0)
28 for j=1:length(v0)
29    vv(1)=v0(j); Jdet(j)=1/2/C0*(v0(j)-m0)^2;% background penalization
30      for i=1:J
31         vv(i+1)=r*vv(i)*(1-vv(i));
32         Phidet(j)=Phidet(j)+1/2/gamma^2*(y(i)-vv(i+1))^2;% misfit
33                   functional
34      end
35     Idet(j)=Phidet(j)+Jdet(j);
36 end
37
38 constant=trapz(v0,exp(-Idet));% approximate normalizing constant
39 P=exp(-Idet)/constant;% normalize posterior distribution
40 prior=normpdf(v0,m0,sqrt(C0));% calculate prior distribution
41
42 figure(1),plot(v0,prior,'k','LineWidth',2)
43 hold on, plot(v0,P,'r--','LineWidth',2), xlabel 'v_0',
44 legend 'prior' J=10^3
```

5.2 Chapter 3 Programs

The programs `p3.m-p7.m`, used to generate the figures in Chapter 3, are presented in this section. Hence various MCMC algorithms used to sample the posterior smoothing distribution are given. Furthermore, optimization algorithms used to obtain solutions of the 4DVAR and w4DVAR variational methods are also introduced. Our general theoretical development of MCMC methods in Section 3.2 employs a notation of u for the state of the chain and w for the proposal. For deterministic dynamics, the state is the initial condition v_0; for stochastic dynamics, it is either the signal v or the pair (v_0, ξ), where ξ is the noise (since this pair determines the signal). Where appropriate, the programs described here use the letter v, and variants thereof, for the state of the Markov chain to keep the connection with the underlying dynamics model.

5.2.1. p3.m

The program `p3.m` contains an implementation of the random walk Metropolis (RWM) MCMC algorithm. The development follows Section 3.2.3, where the algorithm is used to determine the posterior distribution on the initial condition arising from the deterministic logistic map of Example 2.4 given by (5.2). Note that in this case, since the underlying dynamics are deterministic and hence completely determined by the initial condition, the RWM algorithm will provide samples from a probability distribution on $\mathbb{R}$.

As in program `p2.m`, the code is divided into three sections: `setup`, where parameters are defined; `truth`, where the truth and data are generated; and `solution`, where the solution is computed, this time by means of MCMC samples from the posterior smoothing distribution. The parameters in lines 5–10 and the true solution (here taken as only the initial condition, rather than the trajectory it gives rise to) `vt` in line 14 are taken to be the same as those used to generate Figure 2.13. The temporary vector `vv` generated in line 19 is the trajectory corresponding to the truth (`vv(1)=vt` in line 14) and used to calculate the observations `y` in line 20. The true value `vt` will also be used as the initial sample in the Markov chain for this and for all subsequent MCMC programs. This scenario is, of course, impossible in the case that the data is not simulated. However, it is useful when the data is simulated, as it is here, because it can reduce the burn-in time, i.e., the time necessary for the current sample in the chain to reach the target distribution, or the high-probability region of the state space. Because we initialize the Markov chain at the truth, the value of $\mathsf{I}_{\mathrm{det}}(v^\dagger)$, denoted by the temporary variable `Idet`, is required to determine the initial acceptance probability, as described below. It is computed in lines 15–23 exactly as in lines 25–34 of program `p2.m`, as described around equation (5.4).

In the `solution` section, some additional MCMC parameters are defined. In line 28, the number of samples is set to `N` $=10^5$. For the parameters and specific data used here, this is sufficient for convergence of the Markov chain. In line 30, the step-size parameter `beta` is preset such that the algorithm for this particular posterior distribution has a reasonable acceptance probability, or ratio of accepted to rejected moves. A general rule of thumb for this is that it should be somewhere around 0.5, to ensure that the algorithm is not too correlated because of high rejection rate (acceptance probability near zero) and that it is not too correlated because of small moves (acceptance probability near one). The vector `V` defined in line 29 will save all of the samples. This is an example in which preallocation is *very* important. Try using the commands `tic` and `toc` before and respectively after the loop

in lines 33–50 in order to time the chain both with and without preallocation.[1] In line 34, a move is proposed according to the proposal equation (3.15):

$$w^{(k)} = v^{(k-1)} + \beta\iota^{(k-1)},$$

where v(v) is the current state of the chain (initially taken to be equal to the true initial condition v_0), $\iota^{(k-1)}$=randn is an i.i.d. standard normal, and w represents $w^{(k)}$. Indices are not used for v and w because they will be overwritten at each iteration.

The temporary variable vv is again used for the trajectory corresponding to $w^{(k)}$ as a vehicle to compute the value of the proposed $\mathsf{I}_{\det}(w^{(k)}; y)$, denoted in line 42 by `I0prop = J0prop + Phiprop`. In lines 44–46, the decision to accept or reject the proposal is made based on the acceptance probability

$$a(v^{(k-1)}, w^{(k)}) = 1 \wedge \exp(\mathsf{I}_{\det}(v^{(k-1)}; y) - \mathsf{I}_{\det}(w^{(k)}; y)).$$

In practice, this corresponds to drawing a uniform random number rand and replacing v and Idet in line 45 with w and I0prop if `rand<exp(I0-I0prop)` in line 44. The variable bb is incremented if the proposal is accepted, so that the running ratio of accepted moves bb to total steps n can be computed in line 47. This approximates the average acceptance probability. The current sample $v^{(k)}$ is stored in line 48. Notice that here one could replace v by `V(n-1)` in line 34, and by `V(n)` in line 45, thereby eliminating v, and letting w be the only temporary variable. However, the present construction is favorable, because as mentioned above, in general one may not wish to save every sample.

The samples V are used in lines 51–53 to visualize the posterior distribution. In particular, bins of width dx are defined in line 51, and the command hist is used in line 52. The assignment `Z = hist(V,v0)` means first that the real number line is split into M bins with centers defined according to `v0(i)` for $i = 1, \ldots, M$, with the first and last bins corresponding to the negative, respectively positive, half-lines. Second, `Z(i)` counts the number of k for which `V(k)` is in the bin with center determined by `v0(i)`. Again, trapz (5.6) is used to compute the normalizing constant in line 53, directly within the plotting command. The choice of the location of the histogram bins allows for a direct comparison with the posterior distribution calculated from the program p2.m by directly evaluating $\mathsf{I}_{\det}(v; y)$ defined in (5.4) for different values of initial conditions v. This output is then compared with the corresponding output of p2.m for the same parameters in Figure 3.2.

[1] In practice, one may often choose to collect certain statistics from the chain "on the fly" rather than saving every sample, particularly if the state space is high-dimensional and the memory required for each sample is large.

```
clear; set(0,'defaultaxesfontsize',20); format long
%%% p3.m MCMC RWM algorithm for logistic map (Ex. 1.4)
%% setup

J=5;% number of steps
r=4;% dynamics determined by alpha
gamma=0.2;% observational noise variance is gamma^2
C0=0.01;% prior initial condition variance
m0=0.5;% prior initial condition mean
sd=10;rng(sd);% choose random number seed

%% truth

vt=0.3;vv(1)=vt;% truth initial condition
Jdet=1/2/C0*(vt-m0)^2;% background penalization
Phidet=0;% initialization model-data misfit functional
for j=1:J
    % can be replaced by Psi for each problem
    vv(j+1)=r*vv(j)*(1-vv(j));% create truth
    y(j)=vv(j+1)+gamma*randn;% create data
    Phidet=Phidet+1/2/gamma^2*(y(j)-vv(j+1))^2;% misfit functional
end
Idet=Jdet+Phidet;% compute log posterior of the truth

%% solution
% Markov Chain Monte Carlo: N forward steps of the
% Markov Chain on R (with truth initial condition)
N=1e5;% number of samples
V=zeros(N,1);% preallocate space to save time
beta=0.05;% step-size of random walker
v=vt;% truth initial condition (or else update I0)
n=1; bb=0; rat(1)=0;
while n<=N
    w=v+sqrt(2*beta)*randn;% propose sample from random walker
    vv(1)=w;
    Jdetprop=1/2/C0*(w-m0)^2;% background penalization
    Phidetprop=0;
    for i=1:J
            vv(i+1)=r*vv(i)*(1-vv(i));
            Phidetprop=Phidetprop+1/2/gamma^2*(y(i)-vv(i+1))^2;
    end
    Idetprop=Jdetprop+Phidetprop;% compute log posterior of the proposal

    if rand<exp(Idet-Idetprop)% accept or reject proposed sample
        v=w; Idet=Idetprop; bb=bb+1;% update the Markov chain
    end
    rat(n)=bb/n;% running rate of acceptance
    V(n)=v;% store the chain
    n=n+1;
end
dx=0.0005; v0=[0.01:dx:0.99];
Z=hist(V,v0);% construct the posterior histogram
figure(1), plot(v0,Z/trapz(v0,Z),'k','Linewidth',2)% visualize the
  posterior
```

5.2.2. p4.m

The program `p4.m` contains an implementation of the independence dynamics sampler for stochastic dynamics, as introduced in Section 3.2.4. Thus the posterior distribution is on the entire signal $\{v_j\}_{j\in\mathbb{J}}$. The forward model in this case is from Example 2.3, given by (5.1). The smoothing distribution $\mathbb{P}(v|Y)$ is therefore over the state space $\mathbb{R}^{J+1}$.

The sections `setup`, `truth`, and `solution` are defined as for program `p3.m`, but note that now the smoothing distribution is over the entire path, not just over the initial condition, because we are considering stochastic dynamics. Since the state space is now the path space, rather than the initial condition as it was in program `p3.m`, the truth `vt`$\in \mathbb{R}^{J+1}$ is now a vector. Its initial condition is taken as a draw from $N(m_0, C_0)$ in line 16, and the trajectory is computed in line 20, so that at the end, `vt`$\sim \rho_0$. As in program `p3.m`, $v^\dagger$ (`vt`) will be the chosen initial condition in the Markov chain (to ameliorate burn-in issues), and so $\Phi(v^\dagger; y)$ is computed in line 23. Recall from Section 3.2.4 that only $\Phi(\cdot; y)$ is required to compute the acceptance probability in this algorithm.

Notice that the collection of samples `V`$\in \mathbb{R}^{N\times J+1}$ preallocated in line 30 is substantial in this case, illustrating the memory issue that arises when the dimension of the signal space, and number of samples, increases.

The current state of the chain $v^{(k)}$ and the value of $\Phi(v^{(k)}; y)$ are again denoted by `v` and `Phi`, while the proposal $w^{(k)}$ and the value of $\Phi(w^{(k)}; y)$ are again denoted by `w` and `Phiprop`, as in program `p3`. As discussed in Section 3.2.4, the proposal $w^{(k)}$ is an independent sample from the prior distribution ρ_0, similarly to $v^\dagger$, and it is constructed in lines 34–39. The acceptance probability used in line 40 is now

$$a(v^{(k-1)}, w^{(k)}) = 1 \wedge \exp(\Phi(v^{(k-1)}; y) - \Phi(w^{(k)}; y)). \tag{5.7}$$

The remainder of the program is structurally the same as `p3.m`. The outputs of this program are used to plot Figures 3.3, 3.4, and 3.5. Note that in the case of Figure 3.5, we have used $N = 10^8$ samples.

```
clear; set(0,'defaultaxesfontsize',20); format long
%%% p4.m MCMC INDEPENDENCE DYNAMICS SAMPLER algorithm
%%% for sin map (Ex. 1.3) with noise
%% setup

J=10;% number of steps
alpha=2.5;% dynamics determined by alpha
gamma=1;% observational noise variance is gamma^2
sigma=1;% dynamics noise variance is sigma^2
C0=1;% prior initial condition variance
m0=0;% prior initial condition mean
sd=0;rng(sd);% choose random number seed

%% truth

vt(1)=m0+sqrt(C0)*randn;% truth initial condition
Phi=0;

for j=1:J
    vt(j+1)=alpha*sin(vt(j))+sigma*randn;% create truth
    y(j)=vt(j+1)+gamma*randn;% create data
    % calculate log likelihood of truth, Phi(v;y) from (1.11)
    Phi=Phi+1/2/gamma^2*(y(j)-vt(j+1))^2;
end

%% solution
% Markov Chain Monte Carlo: N forward steps of the
% Markov Chain on R^{J+1} with truth initial condition
N=1e5;% number of samples
V=zeros(N,J+1);% preallocate space to save time
v=vt;% truth initial condition (or else update Phi)
n=1; bb=0; rat(1)=0;
while n<=N
    w(1)=sqrt(C0)*randn;% propose sample from the  prior
    Phiprop=0;
    for j=1:J
    w(j+1)=alpha*sin(w(j))+sigma*randn;% propose sample from the prior
    Phiprop=Phiprop+1/2/gamma^2*(y(j)-w(j+1))^2;% compute likelihood
    end
    if rand<exp(Phi-Phiprop)% accept or reject proposed sample
       v=w; Phi=Phiprop; bb=bb+1;% update the Markov chain
    end
   rat(n)=bb/n;% running rate of acceptance
   V(n,:)=v;% store the chain
   n=n+1;
end
% plot acceptance ratio and cumulative sample mean
figure;plot(rat)
figure;plot(cumsum(V(1:N,1))./[1:N]')
xlabel('samples N')
ylabel('(1/N) \Sigma_{n=1}^N v_0^{(n)}')
```

5.2.3. p5.m

The independence dynamics sampler of Section 5.2.2 may be very inefficient, since typical random draws from the dynamics may be unlikely to fit the data as well as the current state, and will then be rejected. The fifth program, `p5.m`, gives an implementation of the pCN algorithm from Section 3.2.4 that is designed to overcome this issue by including the parameter β, which, if chosen small, allows for incremental steps in signal space and hence the possibility of nonnegligible acceptance probabilities. This program is used to generate Figure 3.6

This program is almost identical to `p4.m`, and so only the points at which it differs will be described. First, since the acceptance probability is given by

$$a(v^{(k-1)}, w^{(k)}) = 1 \wedge \exp(\Phi(v^{(k-1)}; y) - \Phi(w^{(k)}; y) + G(v^{(k-1)}) - G(w^{(k)})),$$

the quantity

$$G(u) = \sum_{j=0}^{J-1} \Big(\frac{1}{2} |\Sigma^{-\frac{1}{2}} \Psi(u_j)|^2 - \langle \Sigma^{-\frac{1}{2}} u_{j+1}, \Sigma^{-\frac{1}{2}} \Psi(u_j) \rangle \Big)$$

will need to be computed, both for $v^{(k)}$ (denoted by `v` in lines 31 and 44), where its value is denoted by `G` ($v^{(0)} = v^\dagger$), as well as for $G(v^\dagger)$ which is computed in line 22), and for $w^{(k)}$ (denoted by `w` in line 36) where its value is denoted by `Gprop` in line 39.

As discussed in Section 3.2.4, the proposal $w^{(k)}$ is given by (3.19):

$$w^{(k)} = m + (1 - \beta^2)^{\frac{1}{2}} (v^{(k-1)} - m) + \beta \iota^{(k-1)}; \tag{5.8}$$

here $\iota^{(k-1)} \sim N(0, C)$ are i.i.d. and denoted by `iota` in line 35. Here C is the covariance of the Gaussian measure π_0 given in Equation (2.24) corresponding to the case of trivial dynamics $\Psi = 0$, and m is the mean of π_0. The value of m is given by `m` in line 33.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p5.m MCMC pCN algorithm for sin map (Ex. 1.3) with noise

%% setup
J=10;% number of steps
alpha=2.5;% dynamics determined by alpha
gamma=1;% observational noise variance is gamma^2
sigma=.1;% dynamics noise variance is sigma^2
C0=1;% prior initial condition variance
m0=0;% prior initial condition mean
sd=0;rng(sd);% Choose random number seed

%% truth

vt(1)=m0+sqrt(C0)*randn;% truth initial condition
G=0;Phi=0;

for j=1:J
    vt(j+1)=alpha*sin(vt(j))+sigma*randn;% create truth
    y(j)=vt(j+1)+gamma*randn;% create data
    % calculate log density from (1.--)
    G=G+1/2/sigma^2*((alpha*sin(vt(j)))^2-2*vt(j+1)*alpha*sin(vt(j)));
    % calculate log likelihood phi(u;y) from (1.11)
    Phi=Phi+1/2/gamma^2*(y(j)-vt(j+1))^2;
end

%% solution
% Markov Chain Monte Carlo: N forward steps
N=1e5;% number of samples
beta=0.02;% step-size of pCN walker
v=vt;% truth initial condition (or update G + Phi)
V=zeros(N,J+1); n=1; bb=0; rat=0;
m=[m0,zeros(1,J)];
while n<=N
    iota=[sqrt(C0)*randn,sigma*randn(1,J)];% Gaussian prior sample
    w=m+sqrt(1-beta^2)*(v-m)+beta*iota;% propose sample from the pCN walker
    Gprop=0;Phiprop=0;
    for j=1:J
        Gprop=Gprop+1/2/sigma^2*((alpha*sin(w(j)))^2-2*w(j+1)*alpha*sin
              (w(j)));
        Phiprop=Phiprop+1/2/gamma^2*(y(j)-w(j+1))^2;
    end

    if rand<exp(Phi-Phiprop+G-Gprop)% accept or reject proposed sample
        v=w;Phi=Phiprop;G=Gprop;bb=bb+1;% update the Markov chain
    end
    rat(n)=bb/n;% running rate of acceptance
    V(n,:)=v;% store the chain
    n=n+1;
end
% plot acceptance ratio and cumulative sample mean
figure;plot(rat)
figure;plot(cumsum(V(1:N,1))./[1:N]')
xlabel('samples N')
ylabel('(1/N) \Sigma_{n=1}^N v_0^{(n)}')
```

5.2.4. p6.m

The pCN dynamics sampler is now introduced as program `p6.m`. The independence dynamics sampler of Section 5.2.2 may be viewed as a special case of this algorithm for proposal variance $\beta = 1$. This proposal combines the benefits of tuning the step size β while still respecting the prior distribution on the dynamics. It does so by sampling the initial condition and noise (v_0, ξ) rather than the path itself, in lines 34 and 35, as given by equation (5.8). However, as opposed to the pCN sampler of the previous section, this variable `w` is now interpreted as a sample of (v_0, ξ) and is therefore fed into the path `vv` itself in line 39. The acceptance probability is the same as that of the independence dynamics sampler (5.7), depending only on `Phi`. If the proposal is accepted, both the forcing `u=w` and the path `v=vv` are updated in line 44. Only the path is saved, as in the previous routines, in line 47.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p6.m MCMC pCN Dynamics algorithm for
%%% sin map (Ex. 1.3) with noise
%% setup

J=10;% number of steps
alpha=2.5;% dynamics determined by alpha
gamma=1;% observational noise variance is gamma^2
sigma=1;% dynamics noise variance is sigma^2
C0=1;% prior initial condition variance
m0=0;% prior initial condition mean
sd=0;rng(sd);% Choose random number seed

%% truth

vt(1)=m0+sqrt(C0)*randn;% truth initial condition
ut(1)=vt(1);
Phi=0;
for j=1:J
    ut(j+1)=sigma*randn;
    vt(j+1)=alpha*sin(vt(j))+ut(j+1);% create truth
    y(j)=vt(j+1)+gamma*randn;% create data
    % calculate log likelihood phi(u;y) from (1.11)
    Phi=Phi+1/2/gamma^2*(y(j)-vt(j+1))^2;
end

%% solution
% Markov Chain Monte Carlo: N forward steps
N=1e5;% number of samples
beta=0.2;% step-size of pCN walker
u=ut;v=vt;% truth initial condition (or update Phi)
V=zeros(N,J+1); n=1; bb=0; rat=0;m=[m0,zeros(1,J)];
while n<=N
    iota=[sqrt(C0)*randn,sigma*randn(1,J)];% Gaussian prior sample
    w=m+sqrt(1-beta^2)*(u-m)+beta*iota;% propose sample from the pCN walker
    vv(1)=w(1);
    Phiprop=0;
    for j=1:J
        vv(j+1)=alpha*sin(vv(j))+w(j+1);% create path
        Phiprop=Phiprop+1/2/gamma^2*(y(j)-vv(j+1))^2;
    end

    if rand<exp(Phi-Phiprop)% accept or reject proposed sample
        u=w;v=vv;Phi=Phiprop;bb=bb+1;% update the Markov chain
    end
       rat(n)=bb/n;% running rate of acceptance
       V(n,:)=v;% store the chain
       n=n+1;
end
% plot acceptance ratio and cumulative sample mean
figure;plot(rat)
figure;plot(cumsum(V(1:N,1))./[1:N]')
xlabel('samples N')
ylabel('(1/N) \Sigma_{n=1}^N v_0^{(n)}')
```

5.2.5. p7.m

The next program, `p7.m`, contains an implementation of the weak constrained variational algorithm w4DVAR discussed in Section 3.3. This program is written as a function, while all previous programs were written as scripts. This choice was made for `p7.m` so that the MATLAB built-in function `fminsearch` can be used for optimization in the `solution` section, and the program can still be self-contained. To use this built-in function, it is necessary to define an *auxiliary* objective function `I` to be optimized. The function `fminsearch` can be used within a script, but the auxiliary function would then have to be written separately, so we cannot avoid functions altogether unless we write the optimization algorithm by hand. We avoid the latter in order not to divert the focus of this text from the data-assimilation problem, and algorithms to solve it, to the problem of how to optimize an objective function.

Again the forward model is that given by Example 2.8, namely (5.1). The `setup` and `truth` sections are similar to the previous programs, except that G, for example, need not be computed here. The auxiliary objective function `I` in this case is $\mathsf{I}(\cdot; y)$ from equation (2.21), given by

$$\mathsf{I}(\cdot; y) = \mathsf{J}(\cdot) + \Phi(\cdot; y), \tag{5.9}$$

where

$$\mathsf{J}(u) := \frac{1}{2}\big|C_0^{-\frac{1}{2}}(u_0 - m_0)\big|^2 + \sum_{j=0}^{J-1} \frac{1}{2}\big|\Sigma^{-\frac{1}{2}}\big(u_{j+1} - \Psi(u_j)\big)\big|^2 \tag{5.10}$$

and

$$\Phi(u; y) = \sum_{j=0}^{J-1} \frac{1}{2}\big|\Gamma^{-\frac{1}{2}}\big(y_{j+1} - h(u_{j+1})\big)\big|^2. \tag{5.11}$$

It is defined in lines 38–45. The auxiliary objective function takes as inputs (`u,y,sigma,gamma, alpha,m0, C0,J`), and gives output `out`$= \mathsf{I}(u; y)$, where $u \in R^{J+1}$ (given all the other parameters in its definition—the issue of identifying the input to be optimized over is discussed also below).

The initial guess for the optimization algorithm `uu` is taken as a standard normal random vector over $\mathbb{R}^{J+1}$ in line 27. In line 24, a standard normal random matrix of size 100^2 is drawn and thrown away. This is so that one can easily change the input, e.g., to `randn(z)` for `z`$\in \mathbb{N}$, and induce different random initial vectors `uu` for the optimization algorithm, while keeping the data fixed by the random number seed `sd` set in line 12. The truth `vt` may be used as initial guess by uncommenting line 28. In particular, if the output of the minimization procedure is different for different initial conditions, then it is possible that the objective function $\mathsf{I}(\cdot; y)$ has multiple minima, and hence the posterior distribution $\mathbb{P}(\cdot|y)$ is multimodal. As we have already seen in Figure 3.8, this is certainly true even in the case of scalar deterministic dynamics, when the underlying map gives rise to a chaotic flow.

The MATLAB optimization function `fminsearch` is called in line 32. The *function handle* command `@(u)I(u, ···)` is used to tell `fminsearch` that the objective function `I` is to be considered a function of `u`, even though it may take other parameter values as well (in this case, `y,sigma,gamma,alpha,m0,C0`, and `J`). The outputs of `fminsearch` are the value `vmap` such that `I(vmap)` is minimum, the value `fval = I(vmap)`, and the `exit flag`, which takes the value 1 if the algorithm has converged. The reader is encouraged to use the `help` command for more details on this and other MATLAB functions used in the notes.

The results of this minimization procedure are plotted in lines 34–35 together with the true value $v^\dagger$ as well as the data y. In Figure 3.9, such results are presented, including two minima that were found with different initial conditions.

```
 1  function this=p7
 2  clear;set(0,'defaultaxesfontsize',20);format long
 3  %%% p7.m weak 4DVAR for sin map (Ex. 1.3)
 4  %% setup
 5
 6  J=5;% number of steps
 7  alpha=2.5;% dynamics determined by alpha
 8  gamma=1e0;% observational noise variance is gamma^2
 9  sigma=1;% dynamics noise variance is sigma^2
10  C0=1;% prior initial condition variance
11  m0=0;% prior initial condition mean
12  sd=1;rng(sd);% choose random number seed
13
14  %% truth
15
16  vt(1)=sqrt(C0)*randn;% truth initial condition
17  for j=1:J
18      vt(j+1)=alpha*sin(vt(j))+sigma*randn;% create truth
19      y(j)=vt(j+1)+gamma*randn;% create data
20  end
21
22  %% solution
23
24      randn(100);% try uncommenting or changing the argument for different
25                 % initial conditions -- if the result is not the same,
26                 % there may be multimodality (e.g. 1 & 100).
27      uu=randn(1,J+1);% initial guess
28      %uu=vt;     % truth initial guess option
29
30  % solve with blackbox
31  % exitflag=1 ==> convergence
32  [vmap,fval,exitflag]=fminsearch(@(u)I(u,y,sigma,gamma,alpha,m0,C0,J),uu)
33
34  figure;plot([0:J],vmap,'Linewidth',2);hold;plot([0:J],vt,'r','Linewidth',2)
35  plot([1:J],y,'g','Linewidth',2);hold;xlabel('j');legend('MAP','truth','y')
36
37  %% auxiliary objective function definition
38  function out=I(u,y,sigma,gamma,alpha,m0,C0,J)
39
40  Phi=0;JJ=1/2/C0*(u(1)-m0)^2;
41  for j=1:J
42      JJ=JJ+1/2/sigma^2*(u(j+1)-alpha*sin(u(j)))^2;
43      Phi=Phi+1/2/gamma^2*(y(j)-u(j+1))^2;
44  end
45  out=Phi+JJ;
```

5.3 Chapter 4 Programs

The programs `p8.m-p15.m`, used to generate the figures in Chapter 4, are presented in this section. Various filtering algorithms used to sample the posterior filtering distribution are given, involving both Gaussian approximation and particle approximation. Since these algorithms are run for very large times (large J), they will be divided into only two sections: `setup`, in which the parameters are defined, and `solution`, in which *both* the truth and observations are generated, *and* the online assimilation of the current observation into the filter solution is performed. The generation of truth can be separated into a `truth` section as in the previous sections, but two loops of length J would be required, and loops are inefficient in MATLAB , so the present format is preferred. The programs in this section are all very similar, and their output is also similar, giving rise to Figures 4.3–4.12. With the exception of `p8.m` and `p9.m`, the forward model is given by Example 2.8 (5.1), and the output is identical, given for `p10.m` through `p15.m` in Figures 4.5–4.7 and 4.8–4.10. Figures 4.11 and 4.12 compare the filters from the other Figures. The program `p8.m` features a two-dimensional linear forward model, and `p9.m` features the forward model from Example 2.9 (5.2). At the end of each program, the outputs are used to plot the mean and the covariance as well as the mean-square error of the filter as functions of the iteration number j.

5.3.1. p8.m

The first filtering program is `p8.m`, which contains an implementation of the Kalman filter applied to Example 2.2,

$$v_{j+1} = Av_j + \xi_j, \quad \text{with} \quad A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix},$$

and observed data given by

$$y_{j+1} = Hv_{j+1} + \eta_{j+1}$$

with $H = (1, 0)$ and Gaussian noise. Thus only the first component of v_j is observed.

The parameters and initial condition are defined in the `setup` section, lines 3–19. The vectors `v`, `m` $\in \mathbb{R}^{N\times J}$, `y`$\in \mathbb{R}^J$, and `c` $\in \mathbb{R}^{N\times N\times J}$ are preallocated to hold the truth, mean, observations, and covariance over the `J` observation times defined in line 5. In particular, notice that the true initial condition is drawn from $N(m_0, C_0)$ in line 16, where $m_0 = 0$ and $C_0 = 1$ are defined in lines 10–11. The initial *estimate* of the distribution is defined in lines 17–18 as $N(m_0, C_0)$, where $m_0 \sim N(0, 100I)$ and $C_0 \leftarrow 100C_0$, so that the code may test the ability of the filter to lock onto the true distribution, asymptotically in j, given a poor initial estimate. That is to say, the values of (m_0, C_0) are *changed* such that the initial condition is *not* drawn from this distribution.

The main `solution` loop then follows in lines 21–34. The truth `v` and the data that are being assimilated `y` are sequentially generated within the loop, in lines 24–25. The filter prediction step, in lines 27–28, consists in computing the predictive mean and covariance $\widehat{m}_j$ and $\widehat{C}_j$ as defined in (4.4) and (4.5) respectively:

$$\widehat{m}_{j+1} = Am_j, \quad \widehat{C}_{j+1} = AC_jA^T + \Sigma.$$

Notice that indices are not used for the transient variables `mhat` and `chat` representing $\widehat{m}_j$ and $\widehat{C}_j$, because they will not be saved from one iteration to the next. In lines 30–33, we implement the analysis formulas for the Kalman filter from Corollary 4.2. In particular, the innovation between the observation of the predicted mean and the actual observation, as introduced in Corollary 4.2, is first computed in line 30,

$$d_j = y_j - H\widehat{m}_j. \tag{5.12}$$

Again `d`, which represents d_j, does not have any index for the same reason as above. Next, the Kalman gain defined in Corollary 4.2 is computed in line 31:

$$K_j = \widehat{C}_j H^T (H\widehat{C}_j H^T + \Gamma)^{-1}. \tag{5.13}$$

Once again, an index j is not used for the transient variable `K` representing K_j. Notice that a "forward slash" `/` is used to compute `B/A=B A`$^{-1}$. This is an internal function of MATLAB that will analyze the matrices `B` and `A` to determine an "optimal" method for inversion, given their structure. The update given in Corollary 4.2 is completed in lines 30–32 with the equations

$$m_j = \widehat{m}_j + K_j d_j \quad \text{and} \quad C_j = (I - K_j H)\widehat{C}_j. \tag{5.14}$$

Finally, in lines 36–50, the outputs of the program are used to plot the mean and the covariance as well as the mean-square error of the filter as functions of the iteration number j, as shown in Figure 4.3.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p8.m Kalman Filter, Ex. 1.2
%% setup

J=1e3;% number of steps
N=2;% dimension of state
I=eye(N);% identity operator
gamma=1;% observational noise variance is gamma^2*I
sigma=1;% dynamics noise variance is sigma^2*I
C0=eye(2);% prior initial condition variance
m0=[0;0];% prior initial condition mean
sd=10;rng(sd);% choose random number seed
A=[0 1;-1 0];% dynamics determined by A

m=zeros(N,J);v=m;y=zeros(J,1);c=zeros(N,N,J);% pre-allocate
v(:,1)=m0+sqrtm(C0)*randn(N,1);% initial truth
m(:,1)=10*randn(N,1);% initial mean/estimate
c(:,:,1)=100*C0;% initial covariance
H=[1,0];% observation operator

%% solution % assimilate!

for j=1:J
    v(:,j+1)=A*v(:,j) + sigma*randn(N,1);% truth
    y(j)=H*v(:,j+1)+gamma*randn;% observation

    mhat=A*m(:,j);% estimator predict
    chat=A*c(:,:,j)*A'+sigma^2*I;% covariance predict

    d=y(j)-H*mhat;% innovation
    K=(chat*H')/(H*chat*H'+gamma^2);% Kalman gain
    m(:,j+1)=mhat+K*d;% estimator update
    c(:,:,j+1)=(I-K*H)*chat;% covariance update
end

figure;js=21;plot([0:js-1],v(2,1:js));hold;plot([0:js-1],m(2,1:js),'m');
plot([0:js-1],m(2,1:js)+reshape(sqrt(c(2,2,1:js)),1,js),'r--');
plot([0:js-1],m(2,1:js)-reshape(sqrt(c(2,2,1:js)),1,js),'r--');
hold;grid;xlabel('iteration, j');
title('Kalman Filter, Ex. 1.2');

figure;plot([0:J],reshape(c(1,1,:)+c(2,2,:),J+1,1));hold
plot([0:J],cumsum(reshape(c(1,1,:)+c(2,2,:),J+1,1))./[1:J+1]','m', ...
'Linewidth',2); grid; hold;xlabel('iteration, j');axis([1 1000 0 50]);
title('Kalman Filter Covariance, Ex. 1.2');

figure;plot([0:J],sum((v-m).^2));hold;
plot([0:J],cumsum(sum((v-m).^2))./[1:J+1],'m','Linewidth',2);grid
hold;xlabel('iteration, j');axis([1 1000 0 50]);
title('Kalman Filter Error, Ex. 1.2')
```

5.3.2. p9.m

The program `p9.m` contains an implementation of the 3DVAR method applied to the chaotic logistic map of Example 2.4 (5.2) for $r = 4$. As in the previous section, the parameters and initial condition are defined in the `setup` section, lines 3–16. In particular, notice that the truth initial condition `v(1)` and initial mean `m(1)` are now initialized in lines 12–13 with a *uniform* random number using the command `rand`, so that they are in the interval $[0, 1]$, where the model is well defined. Indeed, the solution will eventually become unbounded if initial conditions are chosen outside this interval. With this in mind, we set the dynamics noise `sigma = 0` in line 8, i.e., deterministic dynamics, so that the true dynamics themselves do not become unbounded.

The analysis step of 3DVAR consists in minimizing

$$\mathsf{I}_{\text{filter}}(v) = \frac{1}{2}|\Gamma^{-\frac{1}{2}}(y_{j+1} - Hv)|^2 + \frac{1}{2}|\widehat{C}^{-\frac{1}{2}}(v - \Psi(m_j))|^2.$$

In this one-dimensional case, we set $\Gamma = \gamma^2$, $\widehat{C} = \sigma^2$ and define $\eta^2 = \gamma^2/\eta^2$. The stabilization parameter η (`eta`) from Example 4.12 is set in line 14, representing the ratio in uncertainty in the data to that of the model; equivalently, it measures trust in the model over the observations. The choice $\eta = 0$ means that the model is irrelevant in the minimization step (4.12) of 3DVAR, in the observed space—the synchronization filter. Since in the example, the signal space and observation space both have dimension equal to 1, the choice $\eta = 0$ simply corresponds to using only the data. In contrast, the choice $\eta = \infty$ ignores the observations and uses only the model.

The 3DVAR setup gives rise to the constant scalar covariance `C` and resultant constant scalar gain `K`; this should not be confused with the changing K_j in (5.13), temporarily defined by `K` in line 31 of `p8.m`. The main `solution` loop follows in lines 20–33. Up to the different forward model, lines 21–22, 24, 26, and 27 of this program are identical to lines 24–25, 27, 30, and 32 of `p8.m`, described in Section 5.3.1. The only other difference is that the covariance updates are not here because of the constant-covariance assumption underlying the 3DVAR algorithm.

The 3DVAR filter may in principle generate the estimated mean `mhat` outside $[0, 1]$, because of the noise in the data. In order to flag potential unbounded trajectories of the filter, which in principle could arise because of this, an extra stopping criterion is included in lines 29–32. To illustrate this, try setting `sigma`$\neq 0$ in line 8. Then the signal will eventually become unbounded, regardless of how small the noise variance is chosen. In this case, the estimate will surely blow up while tracking the unbounded signal. Otherwise, if η is chosen appropriately so as to stabilize the filter, it is extremely unlikely that the estimate will ever blow up. Finally, similarly to `p8.m`, in the last lines of the program we use the outputs of the program in order to produce Figure 4.4, namely plotting the mean and the covariance as well as the mean-square error of the filter as functions of the iteration number j.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p9.m 3DVAR Filter, deterministic logistic map (Ex. 1.4)
%% setup

J=1e3;% number of steps
r=4;% dynamics determined by r
gamma=1e-1;% observational noise variance is gamma^2
sigma=0;% dynamics noise variance is sigma^2
sd=10;rng(sd);% choose random number seed

m=zeros(J,1);v=m;y=m;% pre-allocate
v(1)=rand;% initial truth, in [0,1]
m(1)=rand;% initial mean/estimate, in [0,1]
eta=2e-1;% stabilization coefficient 0 < eta << 1
C=gamma^2/eta;H=1;% covariance and observation operator
K=(C*H')/(H*C*H'+gamma^2);% Kalman gain

%% solution % assimilate!

for j=1:J
    v(j+1)=r*v(j)*(1-v(j)) + sigma*randn;% truth
    y(j)=H*v(j+1)+gamma*randn;% observation

    mhat=r*m(j)*(1-m(j));% estimator predict

    d=y(j)-H*mhat;% innovation
    m(j+1)=mhat+K*d;% estimator update

    if norm(mhat)>1e5
        disp('blowup!')
        break
    end
end
js=21;% plot truth, mean, standard deviation, observations
figure;plot([0:js-1],v(1:js));hold;plot([0:js-1],m(1:js),'m');
plot([0:js-1],m(1:js)+sqrt(C),'r--');plot([1:js-1],y(1:js-1),'kx');
plot([0:js-1],m(1:js)-sqrt(C),'r--');hold;grid;xlabel('iteration, j');
title('3DVAR Filter, Ex. 1.4')

figure;plot([0:J],C*[0:J].^0);hold
plot([0:J],C*[0:J].^0,'m','Linewidth',2);grid
hold;xlabel('iteration, j');title('3DVAR Filter Covariance, Ex. 1.4');

figure;plot([0:J],(v-m).^2);hold;
plot([0:J],cumsum((v-m).^2)./[1:J+1]','m','Linewidth',2);grid
hold;xlabel('iteration, j');
title('3DVAR Filter Error, Ex. 1.4')
```

5.3.3. p10.m

A variation of program `p9.m` is given by `p10.m`, where the 3DVAR filter is implemented for Example 2.3 given by (5.1). Indeed, the remaining programs of this section will all be for the same example, namely Example 2.3, so this will not be mentioned again. In this case, the initial condition is again taken as a draw from the prior $N(m_0, C_0)$ as in `p7.m`, and the initial mean estimate is again *changed* to $m_0 \sim N(0, 100I)$ so that the code may test the ability of the filter to lock onto the signal given a poor initial estimate. Furthermore, for this problem, there is no need to introduce the stopping criterion present in the case of `p9.m` since the underlying deterministic dynamics are dissipative. The output of this program is shown in Figure 4.5.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p10.m 3DVAR Filter, sin map (Ex. 1.3)
%% setup

J=1e3;% number of steps
alpha=2.5;% dynamics determined by alpha
gamma=1;% observational noise variance is gamma^2
sigma=3e-1;% dynamics noise variance is sigma^2
C0=9e-2;% prior initial condition variance
m0=0;% prior initial condition mean
sd=1;rng(sd);% choose random number seed

m=zeros(J,1);v=m;y=m;% pre-allocate
v(1)=m0+sqrt(C0)*randn;% initial truth
m(1)=10*randn;% initial mean/estimate
eta=2e-1;% stabilization coefficient 0 < eta << 1
c=gamma^2/eta;H=1;% covariance and observation operator
K=(c*H')/(H*c*H'+gamma^2);% Kalman gain

%% solution % assimilate!

for j=1:J
    v(j+1)=alpha*sin(v(j)) + sigma*randn;% truth
    y(j)=H*v(j+1)+gamma*randn;% observation

    mhat=alpha*sin(m(j));% estimator predict

    d=y(j)-H*mhat;% innovation
    m(j+1)=mhat+K*d;% estimator update

end

js=21;% plot truth, mean, standard deviation, observations
figure;plot([0:js-1],v(1:js));hold;plot([0:js-1],m(1:js),'m');
plot([0:js-1],m(1:js)+sqrt(c),'r--');plot([1:js-1],y(1:js-1),'kx');
plot([0:js-1],m(1:js)-sqrt(c),'r--');hold;grid;xlabel('iteration, j');
title('3DVAR Filter, Ex. 1.3')

figure;plot([0:J],c*[0:J].^0);hold
plot([0:J],c*[0:J].^0,'m','Linewidth',2);grid
hold;xlabel('iteration, j');
title('3DVAR Filter Covariance, Ex. 1.3');

figure;plot([0:J],(v-m).^2);hold;
plot([0:J],cumsum((v-m).^2)./[1:J+1]','m','Linewidth',2);grid
hold;xlabel('iteration, j');
title('3DVAR Filter Error, Ex. 1.3')
```

5.3.4. p11.m

The next program is `p11.m`. This program comprises an implementation of the extended Kalman filter.It is very similar in structure to `p8.m`, except with a different forward model. Since the dynamics are scalar, the observation operator is defined by setting `H` to take the value 1 in line 16. The predicting covariance $\widehat{C}_j$ is not independent of the mean, as it is for the linear problem `p8.m`. Instead, as described in Section 4.2.2, it is determined via the *linearization* of the forward map around m_j, in line 26:

$$\widehat{C}_{j+1} = (\alpha \cos(m_j))\, C_j\, (\alpha \cos(m_j))\,.$$

As in `p8.m`, we *change* the prior to a poor initial estimate of the distribution to study whether, and how, the filter locks onto a neighborhood of the true signal, despite poor initialization, for large j. This initialization is in lines 15–16, where $m_0 \sim N(0, 100I)$ and $C_0 \leftarrow 10C_0$. Subsequent filtering programs use an identical initialization, with the same rationale as in this case. We will not state this again. The output of this program is shown in Figure 4.6.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p11.m  Extended Kalman Filter, sin map (Ex. 1.3)
%% setup

J=1e3;% number of steps
alpha=2.5;% dynamics determined by alpha
gamma=1;% observational noise variance is gamma^2
sigma=3e-1;% dynamics noise variance is sigma^2
C0=9e-2;% prior initial condition variance
m0=0;% prior initial condition mean
sd=1;rng(sd);% choose random number seed

m=zeros(J,1);v=m;y=m;c=m;% pre-allocate
v(1)=m0+sqrt(C0)*randn;% initial truth
m(1)=10*randn;% initial mean/estimate
c(1)=10*C0;H=1;% initial covariance and observation operator

%% solution % assimilate!

for j=1:J

    v(j+1)=alpha*sin(v(j)) + sigma*randn;% truth
    y(j)=H*v(j+1)+gamma*randn;% observation

    mhat=alpha*sin(m(j));% estimator predict
    chat=alpha*cos(m(j))*c(j)*alpha*cos(m(j))+sigma^2;% covariance predict

    d=y(j)-H*mhat;% innovation
    K=(chat*H')/(H*chat*H'+gamma^2);% Kalman gain
    m(j+1)=mhat+K*d;% estimator update
    c(j+1)=(1-K*H)*chat;% covariance update

end

js=21;% plot truth, mean, standard deviation, observations
figure;plot([0:js-1],v(1:js));hold;plot([0:js-1],m(1:js),'m');
plot([0:js-1],m(1:js)+sqrt(c(1:js)),'r--');plot([1:js-1],y(1:js-1),'kx');
plot([0:js-1],m(1:js)-sqrt(c(1:js)),'r--');hold;grid;xlabel
  ('iteration, j');
title('ExKF, Ex. 1.3')

figure;plot([0:J],c);hold
plot([0:J],cumsum(c)./[1:J+1]','m','Linewidth',2);grid
hold;xlabel('iteration, j');
title('ExKF Covariance, Ex. 1.3');

figure;plot([0:J],(v-m).^2);hold;
plot([0:J],cumsum((v-m).^2)./[1:J+1]','m','Linewidth',2);grid
hold;xlabel('iteration, j');
title('ExKF Error, Ex. 1.3')
```

5.3.5. p12.m

The program `p12.m` contains an implementation of the ensemble Kalman filter, with perturbed observations, as described in Section 4.2.3.The structure of this program is again very similar to those of `p8.m` and `p11.m`, except now an ensemble of particles, of size `N` defined in line 12, is retained as an approximation of the filtering distribution. The ensemble $\{v^{(n)}\}_{n=1}^N$ represented by the matrix `U` is then constructed out of draws from this Gaussian in line 18, and the mean m_0' is reset to the ensemble sample mean.

In line 27, the predicting ensemble $\{\widehat{v}_j^{(n)}\}_{n=1}^N$ represented by the matrix `Uhat` is computed from a realization of the forward map applied to each ensemble member. This is then used to compute the ensemble sample mean $\widehat{m}_j$ (`mhat`) and covariance $\widehat{C}_j$ (`chat`). There is now an ensemble of "innovations" with a new i.i.d. realization $y_j^{(n)} \sim N(y_j, \Gamma)$ for each ensemble member, computed in line 31 (not to be confused with the actual innovation as defined in equation (5.12)),

$$d_j^{(n)} = y_j^{(n)} - H\widehat{v}_j^{(n)}.$$

The Kalman gain K_j (`K`) is computed using (5.13), very similarly to how it is done in `p8.m` and `p11.m`, and the ensemble of updates is computed in line 33:

$$v_j^{(n)} = \widehat{v}_j^{(n)} + K_j d_j^{(n)}.$$

The output of this program is shown in Figure 4.7. Furthermore, long simulations of length $J = 10^5$ were performed for this and the previous two programs, `p10.m` and `p11.m`, and their errors are compared in Figure 4.11.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p12.m Ensemble Kalman Filter (PO), sin map (Ex. 1.3)
%% setup

J=1e5;% number of steps
alpha=2.5;% dynamics determined by alpha
gamma=1;% observational noise variance is gamma^2
sigma=3e-1;% dynamics noise variance is sigma^2
C0=9e-2;% prior initial condition variance
m0=0;% prior initial condition mean
sd=1;rng(sd);% choose random number seed
N=10;% number of ensemble members

m=zeros(J,1);v=m;y=m;c=m;U=zeros(J,N);% pre-allocate
v(1)=m0+sqrt(C0)*randn;% initial truth
m(1)=10*randn;% initial mean/estimate
c(1)=10*C0;H=1;% initial covariance and observation operator
U(1,:)=m(1)+sqrt(c(1))*randn(1,N);m(1)=sum(U(1,:))/N;% initial ensemble

%% solution % assimilate!

for j=1:J

    v(j+1)=alpha*sin(v(j)) + sigma*randn;% truth
    y(j)=H*v(j+1)+gamma*randn;% observation

    Uhat=alpha*sin(U(j,:))+sigma*randn(1,N);% ensemble predict
    mhat=sum(Uhat)/N;% estimator predict
    chat=(Uhat-mhat)*(Uhat-mhat)'/(N-1);% covariance predict

    d=y(j)+gamma*randn(1,N)-H*Uhat;% innovation
    K=(chat*H')/(H*chat*H'+gamma^2);% Kalman gain
    U(j+1,:)=Uhat+K*d;% ensemble update
    m(j+1)=sum(U(j+1,:))/N;% estimator update
    c(j+1)=(U(j+1,:)-m(j+1))*(U(j+1,:)-m(j+1))'/(N-1);% covariance update

end

js=21;% plot truth, mean, standard deviation, observations
figure;plot([0:js-1],v(1:js));hold;plot([0:js-1],m(1:js),'m');
plot([0:js-1],m(1:js)+sqrt(c(1:js)),'r--');plot([1:js-1],y(1:js-1),'kx');
plot([0:js-1],m(1:js)-sqrt(c(1:js)),'r--');hold;grid;xlabel
  ('iteration, j');
title('EnKF, Ex. 1.3')

figure;plot([0:J],c);hold
plot([0:J],cumsum(c)./[1:J+1]','m','Linewidth',2);grid
hold;xlabel('iteration, j');
title('EnKF Covariance, Ex. 1.3');

figure;plot([0:J],(v-m).^2);hold;
plot([0:J],cumsum((v-m).^2)./[1:J+1]','m','Linewidth',2);grid
hold;xlabel('iteration, j');
title('EnKF Error, Ex. 1.3')
```

5.3.6. p13.m

The program `p13.m` contains a particular square-root filter implementation of the ensemble Kalman filter, namely the ETKF filter, described in detail in Section 4.2.4. The program thus is very similar to `p12.m` for the EnKF with perturbed observations. In particular, the filtering distribution of the state is again approximated by an ensemble of particles. The predicting ensemble $\{\widehat{v}_j^{(n)}\}_{n=1}^N$ (`Uhat`), mean $\widehat{m}_j$(`mhat`), and covariance $\widehat{C}_j$ (`chat`) are computed exactly as in `p12.m`. However, this time the covariance is kept in factorized form $\widehat{X}_j\widehat{X}_j^\top = \widehat{C}_j$ in lines 29–30, with factors denoted by `Xhat`. The transformation matrix is computed in line 31,

$$T_j = \left(I_N + \widehat{X}_j^\top H^\top \Gamma^{-1} H \widehat{X}_j\right)^{-\frac{1}{2}},$$

and $X_j = \widehat{X}_j T_j$ (`X`) is computed in line 32, from which the covariance $C_j = X_j X_j^\top$ is reconstructed in line 38. A single innovationd_j is computed in line 34, and a single updated mean m_j is then computed in line 36 using the Kalman gain K_j (5.13) computed in line 35. This is the same as in the Kalman filter and extended Kalman filter (ExKF) of `p8.m` and `p11.m`, in contrast to the EnKF with perturbed observations appearing in `p12.m`. The ensemble is then updated to `U` in line 37 using the formula

$$v_j^{(n)} = m_j + X_j^{(n)}\sqrt{N-1},$$

where $X_j^{(n)}$ is the nth column of X_j.

Notice that the operator that is factorized and inverted has dimension N, which in this case is large in comparison to the state and observation dimensions. This is, of course, natural for computing sample statistics, but in the context of the one-dimensional examples considered here, it makes `p13.m` run far more slowly than `p12.m`. However, in many applications, the signal state-space dimension is the largest, with the observation dimension coming next, and the ensemble size being far smaller than either of these. In this context, the ETKF has become a very popular method. So its relative inefficiency, compared, for example, with the perturbed observations Kalman filter, should not be given too much weight in the overall evaluation of the method. Results illustrating the algorithm are shown in Figure 4.8.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p13.m Ensemble Kalman Filter (ETKF), sin map (Ex. 1.3)
%% setup

J=1e3;% number of steps
alpha=2.5;% dynamics determined by alpha
gamma=1;% observational noise variance is gamma^2
sigma=3e-1;% dynamics noise variance is sigma^2
C0=9e-2;% prior initial condition variance
m0=0;% prior initial condition mean
sd=1;rng(sd);% choose random number seed
N=10;% number of ensemble members

m=zeros(J,1);v=m;y=m;c=m;U=zeros(J,N);% pre-allocate
v(1)=m0+sqrt(C0)*randn;% initial truth
m(1)=10*randn;% initial mean/estimate
c(1)=10*C0;H=1;% initial covariance and observation operator
U(1,:)=m(1)+sqrt(c(1))*randn(1,N);m(1)=sum(U(1,:))/N;% initial ensemble

%% solution % assimilate!

for j=1:J

    v(j+1)=alpha*sin(v(j)) + sigma*randn;% truth
    y(j)=H*v(j+1)+gamma*randn;% observation

    Uhat=alpha*sin(U(j,:))+sigma*randn(1,N);% ensemble predict
    mhat=sum(Uhat)/N;% estimator predict
    Xhat=(Uhat-mhat)/sqrt(N-1);% centered ensemble
    chat=Xhat*Xhat';% covariance predict
    T=sqrtm(inv(eye(N)+Xhat'*H'*H*Xhat/gamma^2));% right-hand sqrt
      transform
    X=Xhat*T;% transformed centered ensemble

    d=y(j)-H*mhat;randn(1,N);% innovation
    K=(chat*H')/(H*chat*H'+gamma^2);% Kalman gain
    m(j+1)=mhat+K*d;% estimator update
    U(j+1,:)=m(j+1)+X*sqrt(N-1);% ensemble update
    c(j+1)=X*X';% covariance update

end

js=21;% plot truth, mean, standard deviation, observations
figure;plot([0:js-1],v(1:js));hold;plot([0:js-1],m(1:js),'m');
plot([0:js-1],m(1:js)+sqrt(c(1:js)),'r--');plot([1:js-1],y(1:js-1),'kx');
plot([0:js-1],m(1:js)-sqrt(c(1:js)),'r--');hold;grid;xlabel
  ('iteration, j');
title('EnKF(ETKF), Ex. 1.3');

figure;plot([0:J],(v-m).^2);hold;
plot([0:J],cumsum((v-m).^2)./[1:J+1]','m','Linewidth',2);grid
plot([0:J],cumsum(c)./[1:J+1]','r--','Linewidth',2);
hold;xlabel('iteration, j');
title('EnKF(ETKF) Error, Ex. 1.3')
```

5.3.7. p14.m

The program `p14.m` is an implementation of the standard SIRS filter from Section 4.3.2. The `setup` section is almost identical to the those of the EnKF methods, because those methods also rely on particle approximations of the filtering distribution. However, the particle filters consistently estimate quite general distributions, while the EnKF is provably accurate only for Gaussian distributions. The truth and data generation and ensemble prediction in lines 24–27 are the same as in `p12.m` and `p13.m`. The way this prediction in line 27 is phrased in Section 4.3.2 is $\widehat{v}_{j+1}^{(n)} \sim \mathbb{P}(\cdot|v_j^{(n)})$. An ensemble of "innovation" terms $\{d_j^{(n)}\}_{n=1}^N$ is again required, but with all terms using the *same* observation, as computed in line 28. Assuming $w_j^{(n)} = 1/N$, then

$$\widehat{w}_j^{(n)} \propto \mathbb{P}(y_j|v_j^{(n)}) \propto \exp\left\{-\frac{1}{2}\left|d_j^{(n)}\right|_\Gamma^2\right\},$$

where $d_j^{(n)}$ is the innovation of the nth particle, as given in (4.27). The vector of unnormalized weights $\{\widehat{w}_j^{(n)}\}_{n=1}^N$ (`what`) is computed in line 29 and normalized to $\{w_j^{(n)}\}_{n=1}^N$ (`w`) in line 30. Lines 32–39 implement the resampling step. First, the cumulative distribution function of the weights $W \in [0,1]^N$ (`ws`) is computed in line 32. Notice that W has the properties $W_1 = w_j^{(1)}$, $W_n \leq W_{n+1}$, and $W_N = 1$. Then N uniform random numbers $\{u^{(n)}\}_{n=1}^N$ are drawn. For each $u^{(n)}$, let n^* be such that $W_{n^*-1} \leq u^{(n)} < W_{n^*}$. This n^* (`ix`) is found in line 34 using the `find` function, which can identify the first or last element in an array to exceed zero (see `help` file): `ix = find ( ws > rand, 1, 'first' )`. This corresponds to drawing the (n^*)th element from the discrete measure defined by $\{w_j^{(n)}\}_{n=1}^N$. The nth particle $v_j^{(n)}$ (`U(j+1,n)`) is set equal to $\widehat{v}_j^{(n^*)}$ (`Uhat(ix)`) in line 37. The sample mean and covariance are then computed in lines 41–42. The rest of the program follows the others, generating the output displayed in Figure 4.9.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p14.m Particle Filter (SIRS), sin map (Ex. 1.3)
%% setup

J=1e3;% number of steps
alpha=2.5;% dynamics determined by alpha
gamma=1;% observational noise variance is gamma^2
sigma=3e-1;% dynamics noise variance is sigma^2
C0=9e-2;% prior initial condition variance
m0=0;% prior initial condition mean
sd=1;rng(sd);% choose random number seed
N=100;% number of ensemble members

m=zeros(J,1);v=m;y=m;c=m;U=zeros(J,N);% pre-allocate
v(1)=m0+sqrt(C0)*randn;% initial truth
m(1)=10*randn;% initial mean/estimate
c(1)=10*C0;H=1;% initial covariance and observation operator
U(1,:)=m(1)+sqrt(c(1))*randn(1,N);m(1)=sum(U(1,:))/N;% initial ensemble

%% solution % Assimilate!

for j=1:J

    v(j+1)=alpha*sin(v(j)) + sigma*randn;% truth
    y(j)=H*v(j+1)+gamma*randn;% observation

    Uhat=alpha*sin(U(j,:))+sigma*randn(1,N);% ensemble predict
    d=y(j)-H*Uhat;% ensemble innovation
    what=exp(-1/2*(1/gamma^2*d.^2));% weight update
    w=what/sum(what);% normalize predict weights

    ws=cumsum(w);% resample: compute cdf of weights
    for n=1:N
        ix=find(ws>rand,1,'first');% resample: draw rand \sim U[0,1] and
        % find the index of the particle corresponding to the first time
        % the cdf of the weights exceeds rand.
        U(j+1,n)=Uhat(ix);% resample: reset the nth particle to the one
        % with the given index above
    end

    m(j+1)=sum(U(j+1,:))/N;% estimator update
    c(j+1)=(U(j+1,:)-m(j+1))*(U(j+1,:)-m(j+1))'/N;% covariance update

end

js=21;% plot truth, mean, standard deviation, observations
figure;plot([0:js-1],v(1:js));hold;plot([0:js-1],m(1:js),'m');
plot([0:js-1],m(1:js)+sqrt(c(1:js)),'r--');plot([1:js-1],y(1:js-1),'kx');
plot([0:js-1],m(1:js)-sqrt(c(1:js)),'r--');hold;grid;xlabel
 ('iteration, j');
title('Particle Filter (Standard), Ex. 1.3');

figure;plot([0:J],(v-m).^2);hold;
plot([0:J],cumsum((v-m).^2)./[1:J+1]','m','Linewidth',2);grid
hold;xlabel('iteration, j');title('Particle Filter (Standard) Error,
  Ex. 1.3')
```

5.3.8. p15.m

The program `p15.m` is an implementation of the SIRS(OP) algorithm from Section 4.3.3. The `setup` section and truth and observation generation are again the same as in the previous programs. The difference between this program and `p14.m` arises because the importance sampling proposal kernel Q_j with density $\mathbb{P}(v_{j+1}|v_j, y_{j+1})$ is used to propose each $\widehat{v}_{j+1}^{(n)}$ given each particular $v_j^{(n)}$; in particular, Q_j depends on the next data point, whereas the kernel P used in `p14.m` has density $\mathbb{P}(v_{j+1}|v_j)$, which is independent of y_{j+1}.

Observe that if $v_j^{(n)}$ and y_{j+1} are both fixed, then $\mathbb{P}\left(v_{j+1}|v_j^{(n)}, y_{j+1}\right)$ is the density of the Gaussian with mean $m'^{(v)}$ and covariance Σ' given by

$$m'^{(n)} = \Sigma'\left(\Sigma^{-1}\Psi\left(v_j^{(n)}\right) + H^\top \Gamma^{-1} y_{j+1}\right), \quad (\Sigma')^{-1} = \Sigma^{-1} + H^\top \Gamma^{-1} H.$$

Therefore, Σ' (`Sig`) and the ensemble of means $\left\{m'^{(n)}\right\}_{n=1}^N$ (vector `em`) are computed in lines 27 and 28 and used to sample $\widehat{v}_{j+1}^{(n)} \sim N(m'^{(n)}, \Sigma')$ in line 29 for all of $\left\{\widehat{v}_{j+1}^{(n)}\right\}_{n=1}^N$ (`Uhat`).

Now the weights are updated by (4.34) rather than (4.27), i.e., assuming $w_j^{(n)} = 1/N$. Then

$$\widehat{w}_{j+1}^{(n)} \propto \mathbb{P}\left(y_{j+1}|v_j^{(n)}\right) \propto \exp\left\{-\frac{1}{2}\left|y_{j+1} - \Psi\left(v_j^{(n)}\right)\right|_{\Gamma+\Sigma}^2\right\}.$$

This is computed in lines 31–32, using another auxiliary "innovation" vector `d` in line 31. Lines 35–45 are again identical to lines 32–42 of program `p14.m`, performing the resampling step and computing the sample mean and covariance.

The output of this program was used to produce Figure 4.10, similar to the other filtering algorithms. Furthermore, long simulations of length $J = 10^5$ were performed for this and the previous three programs, `p12.m`, `p13.m`, and `p14.m`, and their errors are compared in Figure 4.12, similarly to Figure 4.11, comparing the basic filters `p10.m`, `p11.m`, and `p12.m`.

```
clear;set(0,'defaultaxesfontsize',20);format long
%%% p15.m Particle Filter (SIRS, OP), sin map (Ex. 1.3)
%% setup

J=1e3;% number of steps
alpha=2.5;% dynamics determined by alpha
gamma=1;% observational noise variance is gamma^2
sigma=3e-1;% dynamics noise variance is sigma^2
C0=9e-2;% prior initial condition variance
m0=0;% prior initial condition mean
sd=1;rng(sd);% choose random number seed
N=100;% number of ensemble members

m=zeros(J,1);v=m;y=m;c=m;U=zeros(J,N);% pre-allocate
v(1)=m0+sqrt(C0)*randn;% initial truth
m(1)=10*randn;% initial mean/estimate
c(1)=10*C0;H=1;% initial covariance and observation operator
U(1,:)=m(1)+sqrt(c(1))*randn(1,N);m(1)=sum(U(1,:))/N;% initial ensemble

%% solution % Assimilate!

for j=1:J

    v(j+1)=alpha*sin(v(j)) + sigma*randn;% truth
    y(j)=H*v(j+1)+gamma*randn;% observation

    Sig=inv(inv(sigma^2)+H'*inv(gamma^2)*H);% optimal proposal covariance
    em=Sig*(inv(sigma^2)*alpha*sin(U(j,:))+H'*inv(gamma^2)*y(j));% proposal
       mean
    Uhat=em+sqrt(Sig)*randn(1,N);% ensemble optimally importance sampled

    d=y(j)-H*alpha*sin(U(j,:));% ensemble innovation
    what=exp(-1/2/(sigma^2+gamma^2)*d.^2);% weight update
    w=what/sum(what);% normalize predict weights

    ws=cumsum(w);% resample: compute cdf of weights
    for n=1:N
        ix=find(ws>rand,1,'first');% resample: draw rand \sim U[0,1] and
        % find the index of the particle corresponding to the first time
        % the cdf of the weights exceeds rand.
        U(j+1,n)=Uhat(ix);% resample: reset the nth particle to the one
        % with the given index above
    end

    m(j+1)=sum(U(j+1,:))/N;% estimator update
    c(j+1)=(U(j+1,:)-m(j+1))*(U(j+1,:)-m(j+1))'/N;% covariance update

end

js=21;%plot truth, mean, standard deviation, observations
figure;plot([0:js-1],v(1:js));hold;plot([0:js-1],m(1:js),'m');
plot([0:js-1],m(1:js)+sqrt(c(1:js)),'r--');plot([1:js-1],y(1:js-1),'kx');
plot([0:js-1],m(1:js)-sqrt(c(1:js)),'r--');hold;grid;xlabel
  ('iteration, j');
title('Particle Filter (Optimal), Ex. 1.3');
```

5.4 ODE Programs

The programs `p16.m` and `p17.m` are used to simulate and plot the Lorenz '63 and '96 models from Examples 2.6 and 2.7, respectively. These programs are both MATLAB functions, similar to the program `p7.m` presented in Section 5.2.5. The reason for using functions and not scripts is that the black box MATLAB built-in function `ode45` can be used for the time integration (see `help` page for details regarding this function). Therefore, each has an *auxiliary* function defining the right-hand side of the given ODE, which is passed via a *function handle* to `ode45`.

5.4.1. p16.m

The first of the ODE programs, `p16.m`, integrates the Lorenz '63 model 2.6. The setup section of the program, on lines 4–11, defines the parameters of the model and the initial conditions. In particular, a random Gaussian initial condition is chosen in line 9, and a small perturbation to its first (x) component is introduced in line 10. The trajectories are computed on lines 13–14 using the built-in function `ode45`. Notice that the auxiliary function `lorenz63`, defined on line 29, takes as arguments (t, y), prescribed through the definition of the function handle `@(t,y)`, while (α, b, r) are given as fixed parameters `(a,b,r)`, defining the particular instance of the function. The argument t is intended for defining *nonautonomous* ODEs and is spurious here, since it is an *autonomous* ODE, and therefore, t does not appear on the right-hand side. It is nonetheless included for completeness, and it causes no harm. The Euclidean norm of the error is computed in line 16, and the results are plotted similarly to previous programs in lines 18–25. This program is used to plot Figs. 2.6 and 2.7.

5.4.2. p17.m

The second of the ODE programs, `p17.m`, integrates the `J=40`-dimensional Lorenz '96 model 2.7. This program is almost identical to the previous one, where a small perturbation of the random Gaussian initial condition defined on line 9 is introduced on lines 10–11. The major difference is the function passed to `ode45` on lines 14–15, which now defines the right-hand side of the Lorenz '96 model given by the subfunction `lorenz96` on line 30. Again the system is autonomous, and the spurious t-variable is included for completeness. A few of the 40 degrees of freedom are plotted along with the error in lines 19–27. This program is used to plot Figs. 2.8 and 2.9

```
function this=p16
clear;set(0,'defaultaxesfontsize',20);format long
%%% p16.m Lorenz '63 (Ex. 2.6)
%% setup

a=10;b=8/3;r=28;% define parameters
sd=1;rng(sd);% choose random number seed

initial=randn(3,1);% choose initial condition
initial1=initial + [0.0001;0;0];% choose perturbed initial condition

%% calculate the trajectories with blackbox
[t1,y]=ode45(@(t,y) lorenz63(t,y,a,b,r), [0 100], initial);
[t,y1]=ode45(@(t,y) lorenz63(t,y,a,b,r), t1, initial1);

error=sqrt(sum((y-y1).^2,2));% calculate error

%% plot results

figure(1), semilogy(t,error,'k')
axis([0 100 10^-6 10^2])
set(gca,'YTick',[10^-6 10^-4 10^-2 10^0 10^2])

figure(2), plot(t,y(:,1),'k')
axis([0 100 -20 20])


%% auxiliary dynamics function definition
function rhs=lorenz63(t,y,a,b,r)

rhs(1,1)=a*(y(2)-y(1));
rhs(2,1)=-a*y(1)-y(2)-y(1)*y(3);
rhs(3,1)=y(1)*y(2)-b*y(3)-b*(r+a);
```

```
function this=p17
clear;set(0,'defaultaxesfontsize',20);format long
%%% p17.m Lorenz '96 (Ex. 2.7)
%% setup

J=40;F=8;% define parameters
sd=1;rng(sd);% choose random number seed

initial=randn(J,1);% choose initial condition
initial1=initial;
initial1(1)=initial(1)+0.0001;% choose perturbed initial condition

%% calculate the trajectories with blackbox
[t1,y]=ode45(@(t,y) lorenz96(t,y,F), [0 100], initial);
[t,y1]=ode45(@(t,y) lorenz96(t,y,F), t1, initial1);

error=sqrt(sum((y-y1).^2,2));% calculate error

%% plot results

figure(1), plot(t,y(:,1),'k')
figure(2), plot(y(:,1),y(:,J),'k')
figure(3), plot(y(:,1),y(:,J-1),'k')

figure(4), semilogy(t,error,'k')
axis([0 100 10^-6 10^2])
set(gca,'YTick',[10^-6 10^-4 10^-2 10^0 10^2])

%% auxiliary dynamics function definition
function rhs=lorenz96(t,y,F)

rhs=[y(end);y(1:end-1)].*([y(2:end);y(1)] - ...
    [y(end-1:end);y(1:end-2)]) - y + F*y.^0;
```